NTNU
Institutt for datateknikk
og informasjonsvitenskap

LØSNINGSFORSLAG

Kontinuasjonseksamen i

TDT4102 - Prosedyre- og objektorientert programmering

Torsdag 12. august 2010, 09:00 - 13:00

Kontaktperson under eksamen: Trond Aalberg (97631088)

Eksamensoppgaven er utarbeidet av Trond Aalberg
og kvalitetssikret av Svein Erik Bratsberg

Språkform: Bokmål

Tillatte hjelpmidler: Walter Savitch, Absolute C++ eller Lyle Loudon, C++ Pocket Reference

Sensurfrist: Fredag 3 september.

Generell introduksjon

Les gjennom oppgavetekstene nøye og finn ut hva det spørres om. Noen av oppgavene har lengre forklarende tekst, men dette er for å gi mest mulig presis beskrivelse av hva du skal gjøre.

All kode skal være C++.

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner det nødvendig å gjøre. Hver enkelt oppgave er ikke ment å være mer krevende enn det som er beskrevet.

Oppgavesettet er arbeidskrevende og det er ikke foreventet at alle skal klare alle oppgaver innen tidsfristen. Disponer tiden fornuftig!

Oppgavene teller med den andelen som er angitt i prosent. Den prosentvise uttellingen for hver oppgave kan likevel bli justert ved sensur. De enkelte deloppgaver kan også bli tillagt forskjellig vekt.

Oppgave 1: Grunnleggende programmering (30%)

Se nederst på siden for nyttige tips til noen av deloppgavene.

a) Hva skrives ut av følgende kode:

```
int a = 4;
  int b = 5;
  int c = (a++) + (--b);
  cout << c << endl; // 8

  double d = 2;
  int e = 4;
  double f = d / e;
  cout << f << endl; // 0.5

  int g = 2002;
  int h = 10;
  int i = g % h;
  cout << i << endl;  // 2
```

b) Implementer en funksjon **`int countchar(char str[], char c)`** som kan brukes for å finne ut hvor mange ganger et spesifikt tegn er brukt i en tekst-streng (C-streng).

```
//her kan du bruke pekeraritmetikk som vist under
//eller indeksoperatoren kombinert med en variabel for posisjon
//Det som er viktig er å ha en løkke som avsluttes når vi kommer til '\0'
//(eller bruke strlen(char[]), teste på bokstav og inkrementere en teller
int countchar(char str[], char c){
     int count = 0;
     while (*str != '\0'){
       if (*str++ == c){
         count++;
       }
     }
     return count;
  }
```

c) Implementer en funksjon **`string trim(string s)`** som fjerner blanke tegn i starten og slutten av en tekststreng. Hvis du kaller funksjonen med strengen " *Dette er en test* " som argument skal funksjonen returnere strengen "*Dette er en test*".

```
//Mer at det står blanke tegn (flertall). I forslaget under kopierer vi
den delen av strengen vi er interessert i, men det er også greit å  og
f.eks. bruke stringklassens funksjon erase()

string trim(string s){
  //Finner siste tegn som ikke er whitespace
  int lastpos = s.size() - 1;
  while (s[lastpos] == ' '){
    lastpos--;
  }
  ///Finner første tegn som ikke er whitespace
  int startpos = 0;
  while (s[startpos] == ' '){
    startpos++;
  }
  string temp = "";
  for (int i = startpos; i <= lastpos; i++){
    temp += s[i];
  }
  return temp;
}
```

d) Implementer en funksjon **`vector<string> split(string s, char c)`** som deler opp tekststrengen **`s`** i flere delstrenger med tegnet **`c`** som skilletegn. Hvis du gjør kallet **`split(`**"*en;to;tre*"**`, ';')`** skal du få returnert en vector som inneholder strengene "*en*", "*to*", "*tre*".

```
vector<string> split(string s, char c){
  vector<string> splitted;
  //Les tegn for tegn til du kommer til et skilletegn
  //og legg til i en tempvariabel w
  //Lagre hvis w har flere enn 0 tegn for å unngå tomme strenger
  string w = "";
  for (int i = 0; i < s.size(); i++){
    if (s[i] != c){
      w += s[i];
    }else{
      if (w.size() > 0){
        splitted.push_back(w);
        w = "";
      }
    }
  }
  //Siste delstreng må spesialhåndteres
  if (w.size() > 0){
    splitted.push_back(w);
    w = "";
  }
  return splitted;
}
```

e) Implementer en rekursiv funksjon **`string int2string(int i)`** som gjør om et heltall av datatypen int til en strengrepresentasjon av tallet. Kalles funksjonen med heltallet *124* som argument skal den returnere strengen "*124*". NB! oppgaven skal løses med bruk av rekursjon og du kan IKKE benytte deg av tilsvarende funksjoner som allerede finnes fra før i C/C++ biblioteket.

```
string int2string(int i){
//Her trengs det bruk av modulusoperatoren for å finne enkeltsiffer
if (i > 0){
  char x = i % 10 + 48;
  return int2string(i / 10) + x;
}else{
  return "";
}
}
```

f) Implementer en funksjon **`int string2int(string s)`** som gjør om en strengrepresentasjon av et tall til en int-verdi. Hvis tekststrengen inneholder tegn som ikke er tall skal den kaste et unntak. Du bestemmer selv hva slags unnntakstype som skal brukes. Kalles funksjonen med strengen "*124*" skal den returnere int-verdien 124. Kalles funksjon med strengen "*12K3F*" skal den kaste et unntak.

```
int string2int(string s){
//Her er det viktigste å ha med en test på om det er et tall
//og kaste et unntak hvis så er tilfelle
//Vi kan f.eks. kaste en char som unntak så
//kan vi enkelt finne ut hvilket tegn som ikke er et tall.
  int result = 0;
  for (int i = 0; i < s.size(); i++){
    if (s[i] < 48 || s[i] > 57){
      throw s[i];
    }else{
      result += s[i] - 48;
      if ( i < (s.size() - 1)){
        result = result * 10;
      }
    }
  }
  return result;
}
```

Lengden til en string-variabel (antallet tegn i strengen) kan du finne med medlemsfunksjonen **`size()`**.
Enkelttegn i en string-variabel kan leses med medlemsfunksjonen **`at(int)`** eller ved å bruke indeksoperatoren **`[]`**.
Tegnene '0' - '9' finner du på plassene 48 - 57 i ascii-tabellen og for å konvertere fra int-verdiene 0 - 9 til tilsvarende chartegn kan du bruke **`x + 48`**.
Operatorene **`+`** og **`+=`** kan brukes for å legge et tegn til en string-verdi (konkatenere).

Oppgave 2: Klasser og lenkede lister (40%)

I denne oppgaven skal du implementere klasser for å organisere en liste med musikkfiler (tracks). Disse klassene skal for eksempel kunne brukes i et mediaavspillingsprogram, men i denne oppgaven skal du kun ta for deg funksjonalitet som håndterer innlesing fra ei fil, oppretting av en lenket liste hvor hver node inneholder informasjon om de enkelte tracks, og en funksjon for å bla igjennom lista.

Klassen **`Playlist`** representerer en lenket liste med informasjon om musikkfiler (tracks) hvor nodene er instanser av klassen **`Track`**. Innholdet i lista skal leses fra en tekstfil i konstruktøren til **`Playlist`**. I fila er informasjon om hvert track er lagret som en separat linje på formen "*Artist; Tracktitle; Rating; Trackfilename*". Eksempel på innholdet i en slik fil kan være:

Bob Dylan; Spirit on the water; 4; C:\MyMusic\spiritonthewater.mp3
Coldplay; Vival La Vida; 4; C:\MyMusic\vivalavida.mp3
The The; This Is The Day; 6; C:\MyMusic\thisistheday.mp3
No Doubt; Don't Speak; 5; C:\MyMusic\dontspeak.mp3
Annie Lennox; I Saved The World Today; 4; C:\MyMusic\isavedtheworldtoday.mp3

Oppgaven kan besvares del for del, eller du kan velge å implementere klassene samlet. Les igjennom alle deloppgaver slik at du får en oversikt over hva som skal gjøres og sørg for at det går tydelig frem hvor du svarer på hvilke deloppgaver.

a) Klassen **`Track`** skal brukes for å lagre informasjon om de enkelte musikk-filer og instansene skal være noder i en lenket liste. Hvilke medlemsvariabler vil du definere for klassen og hvilke datatyper velger du? Det skal være mulig å lese de enkelte informasjonselementene Artist, Tracktitle, Rating og Trackfilename med get-funksjoner. Merk at Trackfilename bare er et filnavn og at vi ikke skal gjøre noe som helst med selve innholdet i de enkelte musikkfiler.

```
Her kan det være greit å ha medlemsvariabler av typen string (det er også
relevant å ha rating som int). Det er mest praktisk mht. get-funksjonene
å ha egne variabler for hhv. artist, tracktitle, rating og trackfilename.
Du velger selvsagt selv hvilket navn variablene skal ha.

NB! siden vi skal implementere dette som en lenket liste MÅ du også ha
med en variabel som peker til neste Node (for enkeltlenket liste).

class Track{
     private:
        string artist;
        string title;
        string rating; //kunne også vært int
        string trackfilename;
        Track *next;
}
```

b) Forklar hva som er formålet med en klasses konstruktør og implementer en egnet konstruktør for **`Track`**-klassen.

```
En konstruktør skal sørge for at objektene blir instansiert med gyldig
tilstand. En grei løsning på oppgaven er å lage en konstruktør som setter
verdien på alle variablene. Viktig å sette next til NULL.
//Her bruker vi initialiseringsliste for å sette verdiene
```

```
Track::Track(string artist, string title, string rating, string trackfi-
lename): artist(artist), title(title), rating(rating), trackfilena-
me(trackfilename), next(NULL){};
```

c) Vi ønsker at Track-objekter skal være uforanderlige (immutable) etter at de er instansiert. Dvs. at det kun er klassens egne medlemsfunksjoner som skal kunne endre på verdiene. Hvordan oppnår vi dette?

Ved å medlemsvariablene som private og kun tillate at verdiene settes av konstruktøren. Dvs. ikke ha noen set-funksjoner.

d) For Playlist-klassen skal du implementere konstruktøren **`Playlist::Playlist(string playlistfilename)`** og **`Playlist::add(string trackline)`** samt de medlemsvariablene du har bruk for i denne klassen. Konstruktøren skal lese fila som parameteren navngir linje for linje. Innholdet i fila er som beskrevet i den innledende teksten til denne oppgaven. Hver linje i fila er informasjon om enkelttrack og medlemsfunksjonen add skal brukes for å legge ett og ett track til fortløpende slik at de lagres i samme rekkefølge som i playlistfila. I denne oppgaven kan du godt benytte funksjonene som er beskrevet i oppgave 1 for å splitte og trimme teksten. Du kan med fordel benytte deg av biblioteksfunksjonen **`istream& getline (istream& is, string& str)`** for å lese playlist-fila.

```
//Antar at Playlist har denne deklarasjonen
class Playlist{
private:
  void add(string trackline);
  void insert(Track *t); //hjelpefunksjon for a legge til
  Track *first;          //peker til første node
  Track *last;           //peker til siste node
  Track *current;        //peker som brukes av next-funksjonen
public:
  Playlist(string playlistfile);
  Track* next();
};

Playlist::Playlist(string playlistfile){
   //Intitialiserer medlemsvariabler
  first = NULL;
  last = NULL;
  current = NULL;
  //Åpner fil og sjekker at den lar seg åpne
  ifstream input;
  input.open(playlistfile.c_str());
  if (input.fail()){
    cout << "Problemer med å åpne fila " << endl;
    exit(0); //Avslutter ved feil
  }
  //Løkke som leser filen linje for linje og bruker add
  string s;
  while (getline(input, s)){
    add(s);
  }
  input.close();   //Husk å stenge fila
}
```

```
void Playlist::add(string trackline){
  vector<string> trackinfo = split(trackline, ';');
  //Vi antar at trackline har korrekt format
  Track *t = new Track(trim(trackinfo[0]), trim(trackinfo[1]),
                       trim(trackinfo[2]), trim(trackinfo[3]));
  insert(t);
}

void Playlist::insert(Track *t){
  if (first == NULL){
    //tom liste
    first = last = current = t;
  }else{
    //alle andre tilfeller legger vi til på slutten vha last-pekeren
    last->next = t;
    last = t;
  }
}
```

e) Hvordan kan du deklarere **`Playlist:add()`** slik at det bare er konstruktøren (og andre medlemsfunksjoner i Playlist) som kan benytte denne funksjonen?

Da må funksjonen være deklarert som private

f) I oppgave 2c) har vi bestemt at Track-objekter skal være uforanderlige (immutable). Dette skaper et problem når du skal legge til noder i lista. Hva kan du gjøre for at Playlist-klassen (og bare denne) skal få mulighet til å endre på Track-objekter?

Da må Track ha Playlist som friend. Dvs. du trenger denne linja i deklarasjonen av Track-klassen:

```
friend Playlist;
```

g) Implementer operatoren **`<<`** slik at det er mulig å få skrevet ut track-informasjon i samme format som i inputfila. Hvis du har en instans **`Track t`** skal det være mulig å bare skrive **`cout << *t;`** Hvorfor kan vi ikke implementere denne operatoren som medlem av klassen?

Kan ikke være medlem fordi cout er venstre operand og objektet vi skal skrive ut er høyre operand. I implementasjonen av operatorer som medlemsfunksjoner har vi ikke noe parameter for venstresiden siden dette implisitt er objektet selv (this).

Også for denne operatoren er det en praktisk fordel at den er friend av Track. Det gjør at vi kan lese Track-instansenes medlemsvariabler uten bruk av public get-funksjoner.

```
ostream& operator<<(ostream& out, Track t){
  out << t.artist << "; " << t.title << "; " <<
       t.rating << "; " << t.trackfilename;
  return out;
}
```

h) Implementer en medlemsfunksjon **`Track* Playlist::next()`** . Denne funksjonen skal kunne brukes til å bla i listen. Første gang funksjonen kalles skal den returnere første node i lista, andre gang den kalles skal andre node returneres osv. Hvis lista er tom eller du er kommet til slutten av lista skal funksjonen returnere NULL.

I praksis trenger vi en medlemsvariabel i Playlist som kan peke til noden next skal returneres. Se deklarasjonen av Playlist over. Logikken vi skal implementere er at next() skal returnere en node og etterpå flytte pekere til neste. For å løse dette må vi bruke en temp-variabel i next().

```
Track* Playlist::next(){
   Track *temp = current;
   if (current != NULL){
      current = current->next;
   }
   return temp;
}
```

//Det kan også være greit å ha en medlemsfunksjon for å nullstille pekeren som next() bruker slik at man kan gå igjennom lista flere ganger, men i oppgaven spør vi ikke etter dette.

i) Forklar hvorfor det er mulig å bruke løkka under for å bla seg igjennom alle noder i en liste ved hjelp av next-funksjonen. Her er vi ute etter en forklaring på hvordan betingelses-delen av while-løkka fungerer.

```
Playlist favourites("mylist.txt");
Track *t;
while(t = favourites.next()){
   cout << *t;
}
```

En tilordning vil bestandig returnere verdien som ble tilordnet. Hele uttrykket (t = favourites.next()) vil derfor bestandig returnere verdien som ble tilordnet.

favourites.next() gir NULL når vi er kommet til slutten (og en gyldig peker i alle andre tilfeller) og derfor vil (t = NULL) også returnerer verdien NULL

NULL er det samme som 0 og dette er også "tallverdien" for false. Alle andre tall (i praksis adresser) gir true når de opptrer på plasser hvor det er forventet en bool.

Oppgave 3: Arv (30%)

I denne oppgaven skal du jobbe videre med klassene du laget i oppgave 2. Mens implementasjonen i oppgave 2 var en lenket liste hvor track-objektene var organisert i samme rekkefølge som de ble lest fra fil, skal vi nå bruke arv og virtuelle funksjoner og lage subklasser av **`Playlist`** med endret oppførsel.

Merk at den eneste klassen du skal lage subtyper av er **`Playlist`**. Det er i denne oppgaven greit å modifisere din implementasjon av **`Playlist`** og **`Track`** hvis du mener det er nødvendig for å kunne løse oppgaven. Husk å beskrive og vise hvilke endringer du evt. gjør.

I evalueringen legger vi vekt på at du viser kunnskap om og fornuftig bruk av arv samt at du velger hensiktmessige løsninger (ikke unødvendig komplekse løsninger, minst mulig kode, færrest mulig endringer i Playlist og Track-klassene etc.).

a) Lag en subklasse av **`Playlist`** kalt **`RandomPlaylist`** hvor **`next()`** returnerer Track-objektene i tilfeldig rekkefølge. Du skal i denne oppgaven ikke endre på den faktiske rekkefølgen av den lenkede listen, men kun legge til eller endre på funksjonalitet som har med rekkefølgen **`next()`** returnerer objektene i. Forklar løsningen din og dine valg.

Her er det opp til dere å lage en egen løsning basert på hvordan dere har implementert Playlist. En relevant løsning kan være at RandomPlaylist har en vector<Track *> med peker til alle nodene i tilfeldig rekkefølge. I en slik løsning kan du gjenbruke Playlist-konstruktøren og Playlist::next(). Det er ikke forventet at dere bruker tid på å lage intrikate funksjoner for randomisering, her har vi brukt en biblioteksfunksjon.

```
class RandomPlaylist : public Playlist{
private:
  vector<Track*> tracks;
  int pos;
public:
  RandomPlaylist(string filename);
  Track* next();
};

RandomPlaylist::RandomPlaylist(string filename): Playlist(filename){
  Track *t;
  //Lager oss en vector med pekere til Tracks i tilfeldig rekkefølge
  while (t = Playlist::next()){ //bruker Playlist sin next()
    tracks.push_back(t);
  }
  random_shuffle(tracks.begin(), tracks.end());
  pos = 0;
};

Track* RandomPlaylist::next(){
  if (pos < tracks.size()){
    pos++;
    return tracks[pos - 1];
  }else{
    return NULL;
  }
}
```

b) Lag en subklasse av **`Playlist`** kalt **`RatedPlaylist`** hvor nodene i lista er organisert slik at tracks med høyest rating kommer først. Tracks som har lik rating skal sorteres alfabetisk på artist og tittel. Forklar løsningen din og dine valg.

Her er det en grei løsning å lage en subtype som redefinerer konstruktøren og add-funksjonen (i praksis er det bare insettinglogikken som skal endres). Utfordringen er å få tilgang til supertypens variabler og add-funksjonen som er private, samt Track-klassens variabler og da er det bra om du diskuterer disse endringene. Vi kan sette Playlist sine variabler og add til protected og legge til get og set-funksjoner i Track-klassen. Det er også et alternativ å legge til RatedPlaylist som friend i Track-klassen.

Her er det også greit å definere Playlist::insert som virtuell funksjon og bare redefinere denne (i praksis vil ikke dette fungere siden konstruktøren ikke kan bruke virtuelle kall, men det forventer vi ikke at dere skal vite).

En annen kreativ løsning kan være kalle supertypens konstruktør for så å sortere den lenkede listen i subtypens konstruktør. I prinsippet er det egentlig dårlig objektorientert praksis å endre på dataene som er håndtert av supertypen, men i denne oppgaven er det viktigste å demonstrere-bruk og forståelse av arv som mekanisme.

```
class RatedPlaylist : public Playlist{
public:
  RatedPlaylist(string filename);
private:
  void add(string);
  void insertOrdered(Track *t);
};

RatedPlaylist::RatedPlaylist(string playlistfile){
  //Åpner fil og sjekker at den lar seg åpne
  ifstream input;
  input.open(playlistfile.c_str());
  if (input.fail()){
    cout << "Problemer med å åpne fila " << endl;
    exit(0); //Avslutter ved feil
  }
  //Løkke som leser filen linje for linje og bruker add
  string s;
  while (getline(input, s)){
    add(s);
  }
  //Husk å stenge fila
  input.close();
  current = first;
};
```

```
void RatedPlaylist::add(string trackline){
  vector<string> trackinfo = split(trackline, ';');
  Track *t = new Track(trim(trackinfo[0]), trim(trackinfo[1]),
                       trim(trackinfo[2]), trim(trackinfo[3]));
  insertOrdered(t);
}
void RatedPlaylist::insertOrdered(Track *t){

if (first == NULL){
   //lista er tom
   first = last = t;
} else {
   Track *cur = first;
   Track* prev = NULL;
   while((cur->getNext() != NULL) && (*t < *cur)){
     //leter etter riktig plass
     prev = cur;
     cur = cur->getNext();
   }

   if (prev == NULL && *cur < *t){
     //sette inn i før første node
     t->setNext(cur);
     first = t;

   }else if (cur->getNext() == NULL && t* < *cur){
  //setter inn etter siste node
     cur->setNext(t);
     last = t;

   }else{
     //setter inn mellom to noder
     prev->setNext(t);
     t->setNext(cur);

    }
  }
}
```

I denne løsningen har vi gjort noen endringer i Playlist og Track

I Playlist setter vi first, last og current som protected og lager en ny konstruktør (ellers får vi feilmelding fordi Playlist::Playlist() mangler).

```
Playlist::Playlist(): first(NULL), last(NULL), current(NULL){};
```

I Track har vi lagt til getNext og setNext for å kunne endre på next-variabelen. Vi har også definert en sammenligningsoperator for Track objekter.

```
bool operator<(Track &a, Track &b){
  //sorterer på på rating i synkende rekkefølge
  //og på artist/tittel i stigende rekkefølge
  if (a.getRating() != b.getRating()){
    return a.getRating() < b.getRating();
  }else{
    return (a.getArtist() + a.getTitle()) <
             (b.getArtist() + b.getTitle());
  }
}
```